Managing Console I/O Operations

10

Key Concepts

Streams | Stream classes | Unformatted output | Character-oriented input/output | Line-oriented input/output | Formatted output | Formatting functions | Formatting flags | Manipulators | User-defined manipulators

10.1 Introduction

Every program takes some data as input and generates processed data as output following the familiar input-process-output cycle. It is, therefore, essential to know how to provide the input data and how to present the results in a desired form. We have, in the earlier chapters, used **cin** and **cout** with the operators >> and << for the input and output operations. But we have not so far discussed as to how to control the way the output is printed. C++ supports a rich set of I/O functions and operations to do this. Since these functions use the advanced features of C++ (such as classes, derived classes and virtual functions), we need to know a lot about them before really implementing the C++ I/O operations.

Remember, C++ supports all of C's rich set of I/O functions. We can use any of them in the C++ programs. But we restrained from using them due to two reasons. First, I/O methods in C++ support the concepts of OOP and secondly, I/O methods in C cannot handle the user-defined data types such as class objects.

C++ uses the concept of *stream* and *stream classes* to implement its I/O operations with the console and disk files. We will discuss in this chapter, how stream classes support the console-oriented input-output operations. File-oriented I/O operations will be discussed in the next chapter.

10.2

C++ Streams

The I/O system in C++ is designed to work with a wide variety of devices including terminals, disks, and tape drives. Although each device is very different, the I/O system supplies an interface to the programmer that is independent of the actual device being accessed. This interface is known as *stream.*

A stream is a sequence of bytes. It acts either as a *source* from which the input data can be obtained or as a *destination* to which the output data can be sent. The source stream that provides data to the program is called the *input stream* and the destination stream that receives output from the program is called the *output stream.* In other words, a program *extracts* the bytes from an input stream and *inserts* bytes into an output stream as illustrated in Fig. 10.1.

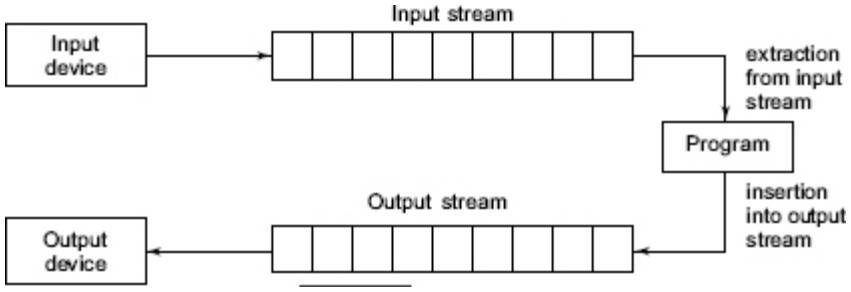


Fig. 10.1 *Data streams*

The data in the input stream can come from the keyboard or any other storage device. Similarly, the data in the output stream can go to the screen or any other storage device. As mentioned earlier, a stream acts as an interface between the program and the input/output device. Therefore, a C++ program handles data (input or output) independent of the devices used.

C++ contains several pre-defined streams that are automatically opened when a program begins its execution. These include cin and cout which have been used very often in our earlier programs. We know that cin represents the input stream connected to the standard

input device (usually the keyboard) and cout represents the output stream connected to the standard output device (usually the screen). Note that the keyboard and the screen are default options. We can redirect streams to other devices or files, if necessary.

10.3 C++ Stream Classes

The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files. These classes are called *stream classes.*Figure 10.2 shows the hierarchy of the stream classes used for input and output operations with the console unit. These classes are declared in the header file *iostream.*This file should be included in all the programs that communicate with the console unit.

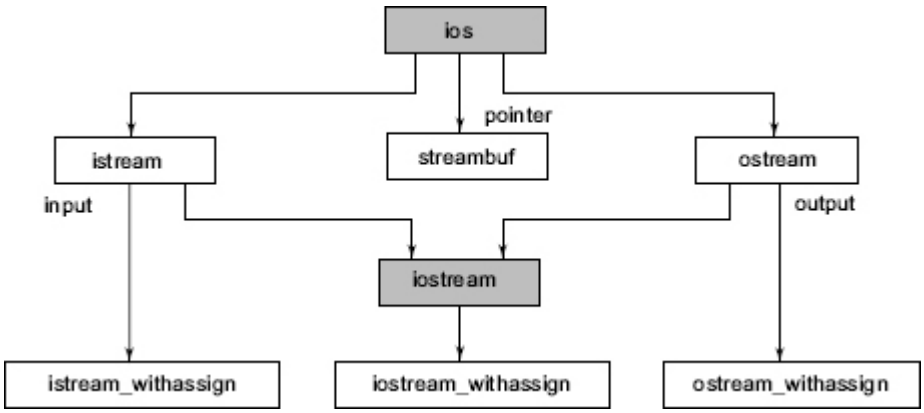


Fig. 10.2 *Stream classes for console I/O operations*

As seen in the Fig. 10.2, **ios** is the base class for **istream** (input stream) and **ostream** (output stream) which are, in turn, base classes for **iostream** (input/output stream). The class **ios** is declared as the virtual base class so that only one copy of its members are inherited by the **iostream.**

The class **ios** provides the basic support for formatted and unformatted I/O operations. The class **istream** provides the facilities

for formatted and unformatted input while the class **ostream** (through inheritance) provides the facilities for formatted output. The class **iostream** provides the facilities for handling both input and output streams. Three classes, namely, **istream_withassign, ostream_ withassign,** and **iostream_withassign** add assignment operators to these classes. Table 10.1 gives the details of these classes.

Table 10.1 *Stream classes for console operations*

Table 10.1 Stream classes for console operations

Class name	Contents
ios (General input/output stream class)	• Contains basic facilities that are used by all other input and output classes • Also contains a pointer to a buffer object (**streambuf** object) • Declares constants and functions that are necessary for handling formatted input and output operations
istream (input stream)	• Inherits the properties of **ios** • Declares input functions such as **get(), getline()** and **read()** • Contains overloaded extraction operator >>
ostream (output stream)	• Inherits the properties of ios • Declares output functions **put()** and **write()** • Contains overloaded insertion operator <<
iostream (input/output stream)	• Inherits the properties of **ios istream** and **ostream** through multiple inheritance and thus contains all the input and output functions
streambuf	• Provides an interface to physical devices through buffers • Acts as a base for **filebuf** class used ios files

10.4 Unformatted I/O Operations

Overloaded Operators >> and <<

We have used the objects **cin** and **cout** (pre-defined in the *iostream*file) for the input and output of data of various types. This has been made possible by overloading the operators >> and << to recognize all the basic C++ types. The >> operator is overloaded in the **istream** class and << is overloaded in the **ostream** class. The following is the general format for reading data from the keyboard:

cin >> variable1 >> variable2 >> .... >> variableN

*variable1, variable2, ...*are valid C++ variable names that have been declared already. This statement will cause the computer to stop the execution and look for input data from the keyboard. The input data for this statement would be:

data1 data2 dataN

The input data are separated by white spaces and should match the type of variable in the **cin** list. Spaces, newlines and tabs will be skipped.

The operator >> reads the data character by character and assigns it to the indicated location. The reading for a variable will be terminated at the encounter of a white space or a character that does not match the destination type. For example, consider the following code:

```
int code;
cin >> code;
```

Suppose the following data is given as input:

4258D

The operator will read the characters upto 8 and the value 4258 is assigned to **code.** The character D remains in the input stream and will be input to the next **cin** statement. The general form for displaying data on the screen is:

```
cout «item1«item2 «....<<itemN
```

The items *item1*through *itemN*may be variables or constants of any basic type. We have used such statements in a number of examples illustrated in previous chapters.

put() and get() Functions

The classes **istream** and **ostream** define two member functions **get()** and **put()** respectively to handle the single character input/output operations . There are two types of **get()** functions. We can use both **get(char *)** and **get(void)** prototypes to fetch a character including the blank space, tab and the newline character. The **get(char *)** version assigns the input character to its argument and the **get(void)** version returns the input character.

Since these functions are members of the input/output stream classes, we must invoke them using an appropriate object.

Example:

```
char c;
cin.get(c); // get a character from keyboard
   // and assign it to c
while(c != '\n')
{
   cout << c;   //  display the character on screen
   cin.get(c);  // get another character
}
```

This code reads and displays a line of text (terminated by a newline character). Remember, the operator >> can also be used to read a character but it will skip the white spaces and newline character. The above **while** loop will not work properly if the statement

```
cin >> c;
```

is used in place of

```
cin.get(c);
```

 **NOTE:** *Try using both of them and compare the results.*

The **get(void)** version is used as follows:

```
. . . . .
char c;
c = cin.get();    //  cin.get(c); replaced
. . . . .
. . . . .
```

The value returned by the function **get()** is assigned to the variable **c.**

The function **put(),** a member of **ostream** class, can be used to output a line of text, character by character. For example,

```
cout.put('x');
```

displays the character **x** and

```
cout.put(ch);
```

displays the value of variable **ch.**

The variable **ch** must contain a character value. We can also use a number as an argument to the function **put().** For example,

```
cout.put(68);
```

displays the character D. This statement will convert the **int** value 68 to a **char** value and display the character whose ASCII value is 68.

The following segment of a program reads a line of text from the keyboard and displays it on the screen.

```
char c;
cin.get(c); // read a character
while(c != '\n')
{
   cout.put(c); //  display the character on screen
   cin.get(c);
}
```

Program 10.1 illustrates the use of these two character handling functions.

Program 10.1 Character I/O with get() and put()

```
#include <iostream>

using namespace std;

int main()
{
int count = 0;
char c;

cout << "INPUT TEXT\n";

cin.get(c);

while(c != '\n')
{
   cout.put(c);
   count++;
   cin.get(c);
```

```
    }
    cout << " \nNumber of characters = " << count << "\n";

    return 0;

    }
```

Input
Object Oriented Programming
Output
Object Oriented Programming
Number of characters = 27

NOTE: *When we type a line of input, the text is sent to the program as soon as we press the RETURN key. The program then reads one character at a time using the statement* ***cin.get(c);*** *and displays it using the statement* ***cout.put(c);.*** *The process is terminated when the newline character is encountered.*

getline() and write() Functions

We can read and display a line of text more efficiently using the line-oriented input/output functions **getline()** and **write().** The **getline()** function reads a whole line of text that ends with a newline character (transmitted by the RETURN key). This function can be invoked by using the object **cin** as follows:

```
cin.getline  (line, size);
```

This function call invokes the function **getline()** which reads character input into the variable line. The reading is terminated as soon as either the newline character '\n' is encountered or size-1 characters are read (whichever occurs first). The newline character

is read but not saved. Instead, it is replaced by the null character. For example, consider the following code:

```
char name[20];
cin.getline(name, 20);
```

Assume that we have given the following input through the keyboard:

```
Bjarne Stroustrup <press RETURN>
```

This input will be read correctly and assigned to the character array **name.** Let us suppose the input is as follows:

```
Object Oriented Programming <press RETURN >
```

In this case, the input will be terminated after reading the following 19 characters:

```
Object Oriented Pro
```

Remember, the two blank spaces contained in the string are also taken into account.

We can also read strings using the operator >> as follows:

```
cin >> name;
```

But remember **cin** can read strings that do not contain white spaces. This means that **cin** can read just one word and not a series of words such as “Bjarne Stroustrup”. But it can read the following string correctly:

```
Bjarne_Stroustrup
```

After reading the string, **cin** automatically adds the terminating null character to the character array.

Program 10.2 demonstrates the use of >> and **getline()** for reading the strings.

Program 10.2 Reading Strings with getline()

```
#include <iostream>

using namespace std;

int main()
{
int size = 20;
char city[20];

cout << "Enter city name: \n";
cin >> city;
cout << "City name:" << city << "\n\n";

cout << "Enter city name again: \n";
cin.getline(city,  size);
cout << "City name now: " << city << "\n\n";

cout << "Enter another city name: \n";
cin.getline(city,  size);
cout << "New city name: " << city << "\n\n";

return 0;
}
```

The output of Program 10.2 would be:

First run
Enter city name:

Delhi
City name: Delhi

Enter city name again:
City name now:
Enter another city name:
Chennai
New city name: Chennai

Second run
Enter city name:
New Delhi
City name: New

Enter city name again:
City name now: Delhi
Enter another city name:
Greater Bombay
New city name: Greater Bombay

NOTE: *During first run, the newline character '\n' at the end of "Delhi" which is waiting in the input queue is read by the* ***getline()*** *that follows immediately and therefore it does not wait for any response to the prompt 'Enter city name again:'. The character '\n' is read as an empty line. During the second run, the word "Delhi" (that was not read by cin) is read by the function* ***getline()*** *and, therefore, here again it does not wait for any input to the prompt 'Enter city name again:'. Note that the line of text "Greater Bombay" is correctly read by the second* ***cin.getline(city,size);*** *statement.*

The **write()** function displays an entire line and has the following form:

cout.write (line, size)

The first argument line represents the name of the string to be displayed and the second argument size indicates the number of characters to display. Note that it does not stop displaying the characters automatically when the null character is encountered. If the size is greater than the length of line, then it displays beyond the bounds of line. Program 10.3 illustrates how **write()** method displays a string.

Program 10.3 Displaying Strings with write()

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
char * string1 = "C++ ";
char * string2 = "Programming";
int m = strlen(string1);
int n = strlen(string2);

for(int i=I; i<n; i++)
{
   cout.write(string2,i);
   cout << "\n";
}
for(i=n; i>0; i--)
{
   cout.write(string2,i);
   cout << "\n";
}
//  concatenating strings
cout.write(string1,m).write(string2,n);
```

```
    cout << "\n";

    //  crossing the boundary
    cout.write(string1,10);

    return 0;
    }
```

The output of Program 10.3 would be:

```
P
Pr
Pro
Prog
Progr
Progra
Program
Programm
Programmi
Programmin
Programming
Programmin
Programmi
Programm
Program
Progra
Progr
Prog
Pro
Pr
P
C++ Programming
C++ Progr
```

The last line of the output indicates that the statement

```
cout.write(string1,  10);
```

displays more characters than what is contained in **string1.**

It is possible to concatenate two strings using the **write()** function. The statement

```
cout.write(string1, m).write(string2, n);
```

is equivalent to the following two statements:

```
cout.write(string1, m);
cout.write(string2, n);
```

10.5 Formatted Console I/O Operations

C++ supports a number of features that could be used for formatting the output. These features include:

- **ios** class functions and flags.
- Manipulators.
- User-defined output functions.

The **ios** class contains a large number of member functions that would help us to format the output in a number of ways. The most important ones among them are listed in Table 10.2.

Table 10.2 *ios format functions*

Table 10.2 *ios format functions*

Function	*Task*
width ()	To specify the required field size for displaying an output value
precision ()	To specify the number of digits to be displayed after the decimal point of a float value
fill()	To specify a character that is used to fill the unused portion of a field
setf()	To specify format flags that can control the form of output display (such as left-justification and right-justification)
unsetf()	To clear the flags specified

Manipulators are special functions that can be included in the I/O statements to alter the format parameters of a stream. Table 10.3 shows some important manipulator functions that are frequently used. To access these manipulators, the file iomanip should be included in the program.

Table 10.3 *Manipulators*

Table 10.3 *Manipulators*

Manipulators	*Equivalent ios function*
setw()	**width()**
setprecision()	**precision()**
setfill()	**fill()**
setiosflags()	**setf()**
resetiosflags()	**unsetf()**

In addition to these functions supported by the C++ library, we can create our own manipulator functions to provide any special output formats. The following sections will provide details of how to use the pre-defined formatting functions and how to create new ones.

Defining Field Width: width()

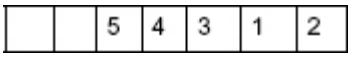
We can use the **width()** function to define the width of a field necessary for the output of an item. Since, it is a member function, we have to use an object to invoke it, as shown below:

```
cout.width(w);
```

where *w*is the field width (number of columns). The output will be printed in a field of *w*characters wide at the right end of the field. The **width()** function can specify the field width for only one item (the item that follows immediately). After printing one item (as per the specifications) it will revert back to the default. For example, the statements

```
cout.width(5);
cout << 543 << 12 << "\n";
```

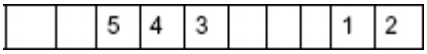
will produce the following output:



The value 543 is printed right-justified in the first five columns. The specification width(5) does not retain the setting for printing the number 12. This can be improved as follows:

```
cout.width(5);
cout << 543;
cout.width(5);
cout << 12 << "\n";
```

This produces the following output:



Remember that the field width should be specified for each item separately. C++ never truncates the values and therefore, if the specified field width is smaller than the size of the value to be printed, C++ expands the field to fit the value. Program 10.4 demonstrates how the function **width()** works.

Program 10.4 Specifying Field Size with width()

```
#include <iostream>
using namespace std;

int main()
{
int items[4] = {10, 8, 12, 15};
int cost[4] = {75, 100, 60, 99};

cout.width(5);
cout << “ITEMS”;
cout.width(8);
cout << “COST”;

cout.width(15);
cout << “TOTAL VALUE” << “\n”;

int sum = 0;

for(int i=0; i<4; i++)
{
   cout.width(5);
   cout << items[i];

   cout.width(8);
   cout << cost[i];

   int value = items[i]  * cost[i];
   cout.width(15);
   cout << value << “\n”;
   sum = sum + value;
}
cout << “\n Grand Total = “;

cout.width(2);
cout << sum << “\n”;
```

```
    return 0;
    }
```

The output of Program 10.4 would be:

ITEMS COST TOTAL VALUE

10 75 750

8 100 800

12 60 720

15 99 1485

Grand Total = 3755

 **NOTE:** *A field of width two has been used for printing the value of sum and the result is not truncated. A good gesture of C++ !*

Setting Precision: precision()

By default, the floating numbers are printed with six digits after the decimal point. However, we can specify the number of digits to be displayed after the decimal point while printing the floating-point numbers. This can be done by using the **precision()** member function as follows:

```
cout.precision(d);
```

where d is the number of digits to the right of the decimal point. For example, the statements

```
cout.precision(3);
cout << sqrt(2) << "\n";
cout << 3.14159 << "\n";
cout << 2.50032 << "\n";
```

will produce the following output:

```
1.141 (truncated)
3.141 (rounded to the nearest cent)
2.5   (no trailing zeros)
```

Not that, unlike the function **width(), precision()** retains the setting in effect until it is reset. That is why we have declared only one statement for the precision setting which is used by all the three outputs.

We can set different values to different precision as follows:

```
cout.precision(3);
cout << sqrt(2) << "\n";
cout.precision(5); // Reset the precision
cout << 3.14159 << "\n";
```

We can also combine the field specification with the precision setting. Example:

```
cout.precision(2);

cout.width(5);

cout << 1.2345;
```

The first two statements instruct: "print two digits after the decimal point in a field of five character width". Thus, the output will be:

Program 10.5 shows how the functions **width()** and **precision()** are jointly used to control the output format.

Program 10.5 Precision Setting with precision()

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
cout << “Precision set to 3 digits \n\n”;
cout.precision(3);

cout.width(10);
cout << “VALUE”;
cout.width(15);
cout << “SQRT_OF_VALUE” << “\n”;

for(int n=i; n<=5; n++)
{

   cout.width(8);
   cout << n;
   cout.width(13);
   cout << sqrt(n) << “\n”;
}
cout << “\n Precision set to 5 digits \n\n”;
cout.precision(5); // precision parameter changed
cout << “ sqrt(i0) = “ << sqrt(i0) << “\n\n”;

cout.precision(0); // precision set to default
cout << “ sqrt(i0) = “ << sqrt(i0) << “ (default setting)\n”;
```

```
    return 0;
    }
```

The output of Program 10.5 would be:

```
  Precision set to 3 digits

VALUE SQRT_OF_VALUE

1 1

2 1.41

3 1.73

4 2

5 2.24

  Precision set to 5 digits
  sqrt(10) = 3.1623
  sqrt(10) = 3.162278  (default setting)
```

NOTE: *Observe the following from the output:*

1. *The output is rounded to the nearest cent (i.e., 1.6666 will be 1.67 for two digit precision but 1.3333 will be 1.33).*
2. *Trailing zeros are truncated.*
3. *Precision setting stays in effect until it is reset.*
4. *Default precision is 6 digits.*

Program 10.6 shows another program demonstrating the functionality of width and precision manipulators:

Program 10.6 Width and Precision Manipulators

```
#include <iostream>

void main()
{
float pi=22.0/7.0;
int i;
cout<<“Value of PI:\n”;
for(i=1;i<=10;i++)
{
   cout.width(i+1);
   cout.precision(i);
   cout<<pi<<“\n”;
}
}
```

The output of Program 10.6 would be:

```
Value of PI:
3.1
3.14
3.143
3.1429
3.14286
3.142857
3.1428571
3.14285707
3.142857075
3.1428570747
```

Filling and Padding: fill()

We have been printing the values using much larger field widths than required by the values. The unused positions of the field are filled with white spaces, by default. However, we can use the fill() function to fill the unused positions by any desired character. It is used in the following form:

```
cout.fill (ch);
```

Where *ch*represents the character which is used for filling the unused positions. Example:

```
cout.fill('*');
cout.width(10);
cout << 5250 << "\n";
```

The output would be:

Financial institutions and banks use this kind of padding while printing cheques so that no one can change the amount easily. Like **precision(), fill()** stays in effect till we change it. See Program 10.7 and its output.

Program 10.7 Padding with Fill()

```
#include <iostream>

using namespace std;

int main( )
{ cout.fill('<');
cout.precision(3);
for(int n=1; n<=6; n++)
```

```
    {
      cout.width(5);
      cout << n;
      cout.width(10);
      cout << 1.0 / float(n) << "\n";
      if (n == 3)
cout.fill  ('>');
    }
    cout << "\nPadding changed \n\n";
    cout.fill  ('#'); //  fill( ) reset
    cout.width (15);
    cout << 12.345678 << "\n";

    return 0;
    }
```

The output of Program 10.7 would be:

```
<<<<1<<<<<<<<<1
<<<<2<<<<<0.5
<<<<3<<<<0.333
>>>4>>>>>>0.25
>>>>5>>>>>>>0.2
>>>>6>>>>>0.167

Padding changed

##########12.346
```

Formatting Flags, Bit-fields and setf()

We have seen that when the function **width()** is used, the value (whether text or number) is printed right-justified in the field width created. But, it is a usual practice to print the text left-justified. How do we get a value printed left-justified? Or, how do we get a floating-point number printed in the scientific notation?

The **setf(),** a member function of the **ios** class, can provide answers to these and many other formatting questions. The **setf()** *(setf*stands for set flags) function can be used as follows:

```
cout.setf(arg1,arg2)
```

The *arg1*is one of the formatting *flags*defined in the class **ios.** The formatting flag specifies the format action required for the output. Another **ios** constant, *arg2,*known as bit *field*specifies the group to which the formatting flag belongs.

Table 10.4 shows the bit fields, flags and their format actions. There are three bit fields and each has a group of format flags which are mutually exclusive. Examples:

```
cout.setf(ios::left, ios::adjustfield);
cout.setf(ios::scientific, ios::floatfield);
```

Table 10.4 *Flags and bit fields for setf() function*

Table 10.4 *Flags and bit fields for setf() function*

Format required	*Flag (arg1)*	*Bit-field (arg2)*
Left-justified output	ios :: left	ios :: adjustfield
Right-justified output	ios :: right	ios :: adjustfield
Padding after sign or base Indicator (like +##20)	ios :: internal	ios:: adjustfield
Scientific notation	ios :: scientific	ios :: floatfield
Fixed point notation	ios :: fixed	ios :: floatfield
Decimal base	ios :: dec	ios :: basefield
Octal base	ios :: oct	ios :: basefield
Hexadecimal base	ios :: hex	ios :: basefield

Note that the first argument should be one of the group members of the second argument.

Consider the following segment of code:

```
cout.fill('*');
cout.setf(ios::left, ios::adjustfield);
```

```
cout.width(15);
cout << "TABLE 1" << "\n";
```

This will produce the following output:

T	A	B	L	E		1	*	*	*	*	*	*	*	*

The statements

```
cout.fill  (' *');
cout.precision(3);
cout.setf(ios::internal, ios::adjustfield);
cout.setf(ios::scientific, ios::floatfield);
cout.width(15);

cout << -12.34567 << "\n";
```

will produce the following output:

-	*	*	*	*	*	1	.	2	3	5	e	+	0	1

NOTE: *The sign is left-justified and the value is right left-justified. The space between them is padded with stars. The value is printed accurate to three decimal places in the scientific notation.*

Program 10.8 demonstrates the manipulation of flag and bit fields for setf() function:

Program 10.8 Manipulation of Flag and Bit Fields

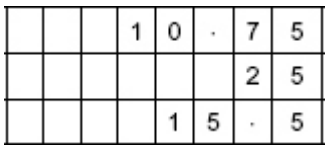
```
#include <iostream>
using namespace std;

void main()
{
int num;
cout<<“Enter an integer value:  “;
cin>>num;

cout<<“The hexadecimal, octal and decimal representation of “<<num<<“ is:  “;

cout.setf(ios::hex, ios::basefield);
cout<<num<<“, ”;

cout.setf(ios::hex, ios::basefield); cout<<num<<“,  “;
cout.setf(ios::oct, ios::basefield); cout<<num<<“ and “;

cout.setf(ios::dec, ios::basefield);


cout<<num<<“ respectively”;
}
```

The output of Program 10.8 would be:

```
Enter an integer value: 92
The hexadecimal, octal and decimal representation of 92 is: 5c, 134 and 92 respectively
```

Displaying Trailing Zeros and Plus Sign

If we print the numbers 10.75, 25.00 and 15.50 using a field width of, say, eight positions, with two digits precision, then the output will be as follows:



Note that the trailing zeros in the second and third items have been truncated.

Certain situations, such as a list of prices of items or the salary statement of employees, require trailing zeros to be shown. The above output would look better if they are printed as follows:

```
10.75
25.00
15.50
```

The **setf()** can be used with the **flag ios::showpoint** as a single argument to achieve this form of output. For example,

```
cout.setf(ios::showpoint); // display trailing zeros
```

would cause cout to display trailing zeros and trailing decimal point. Under default precision, the value 3.25 will be displayed as 3.250000. Remember, the default precision assumes a precision of six digits.

Similarly, a plus sign can be printed before a positive number using the following statement:

```
cout.setf(ios::showpos); //  show +sign
```

For example, the statements

```
cout.setf(ios::showpoint);
cout.setf(ios::showpos);
cout.precision(3);
cout.setf(ios::fixed, ios::floatfield);
cout.setf(ios::internal, ios::adjustfield);
```

```
cout.width(10);
cout << 275.5 << "\n";
```

will produce the following output:

```
+|  |  |2|7|5|.|5|0|0
```

The flags such as **showpoint** and **showpos** do not have any bit fields and therefore are used as single arguments in **setf().** This is possible because the **setf()** has been declared as an overloaded function in the class **ios.** Table 10.5 lists the flags that do not possess a named bit field. These flags are not mutually exclusive and therefore can be set or cleared independently.

Table 10.5 *Flags that do not have bit fields*

Flag	*Meaning*
ios :: showbase	Use base indicator on output
ios :: showpos	Print + before positive numbers
ios :: showpoint	Show trailing decimal point and zeroes
ios :: uppercase	Use uppercase letters for hex output
ios :: skipus	Skip white space on input
ios :: unitbuf	Flush all streams after insertion
ios :: stdio	Flush **stdout** and **stderr** after insertion

Program 10.9 demonstrates the setting of various formatting flags using the overloaded **setf()** function.

Program 10.9 Formatting with Flags in setf()

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{

cout.fill('*');
cout.setf(ios::left, ios::adjustfield);
cout.width(10);
cout << "VALUE";
cout.setf(ios::right, ios::adjustfield);
cout.width(15);
cout << "SQRT OF VALUE" << "\n";

cout.fill('.');
cout.precision(4);
cout.setf(ios::showpoint);
cout.setf(ios::showpos);
cout.setf(ios::fixed, ios::floatfield);

for(int n=1; n<=10; n++)
{
   cout.setf(ios::internal, ios::adjustfield);
   cout.width(5);
   cout << n;

   cout.setf(ios::right, ios::adjustfield);
   cout.width(20);
   cout << sqrt(n) << "\n";

}
   //  floatfield changed
   cout.setf(ios::scientific, ios::floatfield);
   cout << "\nSQRT(100) = " << sqrt(100) << "\n";
```

```
    return 0;
}
```

The output of Program 10.9 would be:

```
VALUE*********SQRT OF VALUE
+...1 +1.0000
+...2 +1.4142
+...3 +1.7321
+...4 +2.0000
+...5 +2.2361
+...6 +2.4495
+...7 +2.6458
+...8 +2.8284
+...9 +3.0000
+..10 +3.1623

SQRT(100)  = +1.0000e+001
```

NOTE:

1. *The flags set by **setf()** remain effective until they are reset or unset.*
2. *A format flag can be reset any number of times in a program.*
3. *We can apply more than one format controls jointly on an output value.*
4. *The setf() sets the specified flags and leaves others unchanged.*

10.6 Managing Output with Manipulators

The header file *iomanip*provides a set of functions called *manipulators*which can be used to manipulate the output formats. They provide the same features as that of the **ios** member functions and flags. Some manipulators are more convenient to use than their counterparts in the class **ios.** For example, two or more manipulators can be used as a chain in one statement as shown below:

```
cout << manip1 << manip2 << manip3 << item;
cout << manip1 << item1 << manip2 << item2;
```

This kind of concatenation is useful when we want to display several columns of output.

The most commonly used manipulators are shown in Table 10.6. The table also gives their meaning and equivalents. To access these manipulators, we must include the file *iomanip*in the program.

Table 10.6 *Manipulators and their meanings*

Table 10.6 *Manipulators and their meanings*

Manipulator	*Meaning*	*Equivalent*
setw (int *w*)		
setprecision(int *d*)	Set the field width to w.	width( )
	Set the floating point precision to *d*.	precision( )
setfill(int *c*)	Set the fill character to c.	fill( )
setiosflags(long *f*)	Set the format flag *f*.	setf( )
reseticsflags(long *f*)	Clear the flag specified by *f*.	unsetf( )
endl	Insert new line and flush stream.	"\n"

Some examples of manipulators are given below:

```
cout << setw(i0) << i2345;
```

This statement prints the value 12345 right-justified in a field width of 10 characters. The output can be made left-justified by modifying the statement as follows:

```
cout << setw(10) << setiosflags(ios::left) << i2345;
```

One statement can be used to format output for two or more values. For example, the statement

```
cout    << setw(5)  << setprecision(2) << 1.2345
<< setw(i0) << setprecision(4) << sqrt(2)
<< setw(i5) << setiosflags(ios::scientific) << sqrt(3);
<< endl;
```

will print all the three values in one line with the field sizes of 5, 10, and 15 respectively. Note that each output is controlled by different sets of format specifications.

We can jointly use the manipulators and the **ios** functions in a program. The following segment of code is valid:

```
cout.setf(ios::showpoint);
```

```
cout.setf(ios::showpos);
cout << setprecision(4);
cout << setiosflags(ios::scientific);
cout << setw(10) << 123.45678;
```

NOTE: *There is a major difference in the way the manipulators are implemented as compared to the **ios** member functions. The **ios** member function return the previous format state which can be used later, if necessary. But the manipulator does not return the previous format state. In case, we need to save the old format states, we must use the **ios** member functions rather than the manipulators. Example:.*

```
cout.precision(2); // previous state
int p = cout.precision(4); // current state;
```

When these statements are executed, **p** will hold the value of 2 (previous state) and the new format state will be 4. We can restore the previous format state as follows:

```
cout.precision(p); // p = 2
```

Program 10.10 illustrates the formatting of the output values using both manipulators and **ios** functions.

Program 10.10 Formatting with Manipulators

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{

   cout.setf(ios::showpoint);

   cout << setw(5) << "n"
<< setw(i5) << "Inverse_of_n"
<< setw(i5) << "Sum_of_terms\n\n";

   double term,  sum = 0;

   for(int n=i; n<=10; n++)
   {

term = 1.0 / float(n);

sum = sum + term;

cout << setw(5) << n

<< setw(i4) << setprecision(4)

<< setiosflags(ios::scientific) << term
```

```
<< setw(i3) << resetiosflags(ios::scientific)

<< sum << endl;

    }
    return 0;
    }
```

The output of Program 10.10 would be:

```
n Inverse_of_n Sum_of_terms
1 1.0000e+000 1.0000
2 5.0000e-001 1.5000
3 3.3333e-001 1.8333
4 2.5000e-001 2.0833
5 2.0000e-001 2.2833
6 1.6667e-001 2.4500
7 1.4286e-001 2.5929
8 1.2500e-001 2.7179
9 1.1111e-001 2.8290
10 1.0000e-001 2.9290
```

Designing Our Own Manipulators

We can design our own manipulators for certain special purposes. The general form for creating a manipulator without any arguments is:

```
ostream & manipulator (ostream & output)
{
. . . . .
. . . . . (code)
. . . . .
return output;
}
```

Here, the *manipulator*is the name of the manipulator under creation. The following function defines a manipulator called **unit** that displays “inches”:

```
ostream  & unit (ostream & output)
{
output << “ inches”;
return output;
}
```

The statement

```
cout << 36 << unit;
```

will produce the following output

```
36 inches
```

We can also create manipulators that could represent a sequence of operations. Example:

```
ostream & show(ostream & output)
{
output.setf(ios::showpoint);
output.setf(ios::showpos);
output << setw(10);
return output;
}
```

This function defines a manipulator called **show** that turns on the flags **showpoint** and **showpos** declared in the class **ios** and sets the field width to 10.

Program 10.11 illustrates the creation and use of the user-defined manipulators. The program creates two manipulators called **currency** and **form** which are used in the **main** program.

Program 10.11 User-Defined Manipulators

```
#include <iostream>
#include <iomanip>

using namespace std;

//  user-defined manipulators

ostream & currency(ostream & output)
{
output << “Rs”;
return output;
}

ostream & form(ostream & output)
{
output.setf(ios::showpos);
output.setf(ios::showpoint);
output.fill(‘*’);
output.precision(2);
output << setiosflags(ios::fixed)
   << setw(10);
return output;
}
int main()
{
cout << currency << form << 7864.5;

return 0;
}
```

The output of Program 10.11 would be:

Rs**+7864.50

Note that **form** represents a complex set of format functions and manipulators.

Summary

- In C++, the I/O system is designed to work with different I/O devices. This I/O system supplies an interface called ‘stream’ to the programmer, which is independent of the actual device being used.
- A stream is a sequence of bytes and serves as a source or destination for an I/O data.
- The source stream that provides data to the program is called the *input stream*and the destination stream that receives output from the program is called the *output stream.*
- The C++ I/O system contains a hierarchy of stream classes used for input and output operations. These classes are declared in the header file **‘iostream’.**
- **cin** represents the input stream connected to the standard input device and **cout** represents the output stream connected to the standard output device.
- The **istream** and **ostream** classes define two member functions **get()** and **put()** to handle the single character I/O operations.
- The >> operator is overloaded in the **istream** class as an extraction operator and the << operator is overloaded in the **ostream** class as an insertion operator.
- We can read and write a line of text more efficiently using the line oriented I/O functions **getline()** and **write()** respectively.

- The ios class contains the member functions such as **width(), precision(), fill(), setf(), unsetf()** to format the output.
- The header file **'iomanip'** provides a set of manipulator functions to manipulate output formats. They provide the same features as that of **ios** class functions.
- We can also design our own manipulators for certain special purposes.

Key Terms

adjustfield | basefield | bit-fields | console I/O operations | decimal base | destination stream | field width | **fill()** | filling | fixed point notation | flags | floatfield | formatted console I/O | formatting flags | formatting functions | **get()** | **getline()** | hexadecimal base | input stream | internal | **ios** | iomanip | iostream | istream | left-justified | manipulator | octal base | ostream | output stream | padding | **precision()** | **put()** | **resetiosflags()** | right-justified | scientific notation | **setf()** | **setfill()** | **setiosflags()** | **setprecision()** | setting precision | **setw()** | showbase | showpoint | showpos | skipus | source stream | standard input device | standard output device | stream classes | streambuf | streams | unitbuf | **unsetf()** | **width()** | **write()**

Review Questions

10.1 What is a stream?

10.2 Describe briefly the features of I/O system supported by C++.

10.3 How do the I/O facilities in C++ differ from that in C?

10.4 Why are the words such as **cin** and **cout** not considered as keywords?

10.5 How is **cout** able to display various types of data without any special instructions?

10.6 Why is it necessary to include the file iostream in all our programs?

10.7 Discuss the various forms of **get()** function supported by the input stream. How are they used?

10.8 How do the following two statements differ in operation?

```
cin >> c;
cin.get(c);
```

10.9 Both **cin** and **getline()** function can be used for reading a string. Comment.

10.10 Discuss the implications of size parameter in the following statement:

```
cout.write(line,  size);
```

10.11 What does the following statement do?
cout.write(s1,m).write(s2,n);

10.12 What role does the **iomanip** file play?

10.13 What is the role of **file()** function? When do we use this function?

10.14 Discuss the syntax of **set()** function.

10.15 What is the basic difference between manipulators and **ios** member functions in implementation? Give examples.

10.16 State whether the following statements are TRUE or FALSE.

(a) A C++ stream is a file.

(b) C++ never truncates data.

(c) The main advantage of **width()** function is that we can use one width specification for more than one items.

(d) The **get(void)** function provides a single-character input that does not skip over the white spaces.

(e) The header file **iomanip** can be used in place of iostream.

(f) We cannot use both the C I/O functions and C++ I/O functions in the same program.

(g) A programmer can define a manipulator that could represent a set of format functions.

10.17 What will be the result of the following programs segment?

```
for(i=0.25; i<=1.0; i=i+0.25)
{
cout.precision(5);
cout.width(7);
cout << i;
cout.width(10);
cout <<i*i<< "\n";
}
cout << setw(10)  << "TOTAL ="
<< setw(20) << setprecision(2) << 1234.567
<< endl;
```

10.18 Discuss the syntax for creating user-defined manipulators. Design a single manipulator to provide the following output specifications for printing float values:

(a) 10 columns width

(b) Right-justified

(c) Two digits precision

(d) Filling of unused places with*

(e) Trailing zeros shown

Debugging Exercises

10.1 To get the output Buffer1: Jack and Jerry Buffer2: Tom and Mono, what do you have to do in the following program?

```
#include <iostream.h>
void main()
{
char buffer1[80];
char buffer2[80];

cout << "Enter value for buffer1 :  ";
cin  >> buffer1;
cout << "Buffer1 : " << buffer1 << endl;

cout << "Enter value for buffer2 :  ";
cin.getline(buffer2,  80);
cout << "Buffer2 : " << buffer2 << endl;

}
```

10.2 Will the statement cout.setf(ios::right) work or not?

```
#include <iostream.h>
void main()
{
cout.width(5);
cout << "99" << endl;

cout.setf(ios::left);
cout.width(5);
cout << "99" << endl;
```

```
cout.setf(ios::right);
cout << “99” << endl;
}
```

10.3 Identify the error in the following program, if any:

```
#include <iostream>
#include <conio.h>
void main()
{
float pi=22.0/7.0;
clrscr();
cout.fill(“$”);
cout.width(10);
cout.precision(2);
cout<<pi<<“\n”;

getch();
}
```

10.4 State errors, if any, in the following statements.

(a) cout << (void*) amount;

(b) cout << put(“John”);

(c) cout << width();

(d) int p = cout.width(10);

(e) cout.width(10).precision(3);

(f) cout.setf(ios::scientific, ios::left);

(g) ch = cin.get();

(h) cin.get().get();

(i) cin.get(c).get();

(j) cout << setw(5) << setprecision(2);

(k) cout << resetiosflags(ios::left |ios::showpos);

Programming Exercises

10.1 Write a program to read a list containing item name, item code, and cost interactively and produce a three column output as shown below.

NAME	*CODE*	*COST*
Turbo C++	1001	250.95
C Primer	905	95.70
.....	...	.....
.....	...	.....
.....	...	.....

Note that the name and code are left-justified and the cost is right-justified with a precision of two digits. Trailing zeros are shown. WEB

10.2 Modify the above program to fill the unused spaces with hyphens.

10.3 Write a program which reads a text from the keyboard and displays the following information on the screen in two columns:

(a) Number of lines

(b) Number of words

(c) Number of characters

Strings should be left-justified and numbers should be right-justified in a suitable field width. WEB

10.4 Write a program to find the value of e in the following series:

$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \cdots\cdots$ upto acc = 0.0001